

Quiz

In a nonempty max-heap, which node is guaranteed to contain a largest key? Select one answer.

- a. The rightmost leaf.
- ☒ b. The root.
- c. The leftmost leaf.

In a singly linked list, what does a node's next pointer store when the node has a successor? Select one answer.

- a. The position of the node in sorted order.
- ☒ b. The memory address of the next node.
- c. The key of the previous node.

Given an upper bound on the list's size, the abstract data type List can be implemented using either an array or a linked list.

↪ meaning that the list will not be longer than C

☒ True

False

(array requires fixed maximum size)

Binary Search looks for 4 in $A = (1, 3, 5, \overline{7}, \overline{9}, 11, 13)$. Which sequence of values does it inspect? Select one answer.

² ³ ⁴
~~1~~ ~~3~~ ~~5~~
~~5~~

- a. (7, 3, 1)
- ☒ b. (7, 3, 5)
- c. (7, 5, 3)

$$\text{Formula } m = \left\lfloor \frac{r+l}{2} \right\rfloor$$

Quicksort has just partitioned an array of distinct integers around a pivot, as in the lecture. For each statement, decide whether it is guaranteed to hold.

Wahr	Falsch		
	✗	The entries to the left of the pivot are already sorted.	
✗		Every entry to the left of the pivot is smaller than every entry to its right. <i>that's exactly what "Aufteilen()" does</i>	
✗		The pivot is in its final sorted position.	

How many comparisons between array entries does the lecture's Merge algorithm perform when merging (1, 3, 4) and (2, 5, 6)? Select one answer.

a. 5

b. 3

☒ c. 4

1 3 4
2 5 6

⇒

3 4
2 5 6

⇒

3 4
5 6

⇒

4
5 6

⇒

B = [1, 2, 3, 4, 5, 6]

↳ B = [1,

↳ B = [1, 2

↳ B = [1, 2, 3

↳ B = [1, 2, 3, 4

During Insertion Sort, the already sorted prefix must contain the smallest entries of the entire array.

True

☒ False

It is only relatively sorted and the elements are not necessarily at their final index

How many nonempty contiguous substrings of the bitstring "010" contain exactly one 1?

Answer:

4

"01", "10", "1", "010"

Bubble Sort starts with $A = (2, 4, 1, 3, 5)$ and makes left-to-right passes. After how many complete passes is the array first sorted?

Answer:

2

after 1 pass: (2, 1, 3, 4, 5)

after 2 passes: (1, 2, 3, 4, 5)

How many integer multiplications does the iterative-squaring algorithm from Homework 3 use to compute a^8 ?

Answer:

$$\begin{aligned} a^2 &= a * a \\ a^4 &= a^2 * a^2 \\ a^8 &= a^4 * a^4 \end{aligned}$$

Let $T(1) = 1$ and $T(n) = T(n/2) + n$ for powers of two $n > 1$. Which tight asymptotic bound holds for $T(n)$? Select one answer.

a. $T(n) = \Theta(n \log n)$.

$$a = 1, b = 1$$

☒ b. $T(n) = \Theta(n)$.

$$\log(a) = \log(1) = 0 < 1 = b$$

c. $T(n) = \Theta(\log n)$.

Master Theorem

$$\Rightarrow T(n) = \Theta(n^b) = \Theta(n)$$

Let n range over integers with $n \geq 5$. For each statement, decide whether it is true or false.

Wahr	Falsch		
	☒	$2^n = \Theta(4^n)$.	$\lim_{n \rightarrow \infty} \frac{2^n}{4^n} = \lim_{n \rightarrow \infty} \left(\frac{2}{4}\right)^n = 0$
☒		$\log_2 n = \Theta(\log_4 n)$.	basis of log doesn't matter
☒		$2^{n+1} = \Theta(2^n)$.	$2^{n+1} = 2 \cdot 2^n = \Theta(2^n)$

(sehr schwierig)

Let $n = 2^k$ for an integer $k \geq 2$. How many calls to f does the following code make? Select one answer.

```
i ← 1
while i ≤ n :
  j ← 1
  while j ≤ i :
    f()
  j ← 2j
i ← 2i
```

a. $\frac{k(k+1)}{2}$

b. 2^k

☒ c. $\frac{(k+1)(k+2)}{2}$

d. $2^{k+1} - 1$

für $k=2$:

$$\frac{2 \cdot 3}{2} = 3$$

$$2^2 = 4$$

$$\frac{3 \cdot 4}{2} = 6$$

$$2^3 - 1 = 7$$

Just try $k=2$. $n = 2^k = 2^2 = 4$

Iteration 1 = 1

$$\begin{aligned} j=1 &\leq 1 &\Rightarrow 1 \text{ call} \\ j=2 &\not\leq 1 \end{aligned}$$

Iteration 1 = 2

$$\begin{aligned} j=1 &\leq 2 &\Rightarrow 1 \text{ call} \\ j=2 &\leq 2 &\Rightarrow 1 \text{ call} \\ j=4 &\not\leq 2 \end{aligned}$$

Iteration 1 = 4

$$\begin{aligned} j=1 &\leq 4 &\Rightarrow 1 \text{ call} \\ j=2 &\leq 4 &\Rightarrow 1 \text{ call} \\ j=4 &\leq 4 &\Rightarrow 1 \text{ call} \\ j=8 &\not\leq 4 \end{aligned}$$

Total 6 calls für $k=2$

$$\sum_{i=1}^{k+1} \sum_{j=1}^i 1$$

(schwierig zu sehen)

$$= \sum_{i=1}^{k+1} i$$

$$= \frac{(k+1)(k+2)}{2}$$

Gauss
Summenformel

Discussion of Exercise Sheet 4

Exercise 4.4 One-Looped Sort (2 bonus points). → Peergrading Aufgabe

Consider the following pseudocode whose goal is to sort an array A containing n integers.

Algorithm 1 Input: array $A[0 \dots n-1]$.

```

 $i \leftarrow 0$ 
while  $i < n$  do
  if  $i = 0$  or  $A[i] \geq A[i-1]$  then:
     $i \leftarrow i + 1$ 
  else
    swap  $A[i]$  and  $A[i-1]$ 
     $i \leftarrow i - 1$ 

```

- (b) Explain why the algorithm correctly sorts any input array. Formulate a reasonable loop invariant, prove it (e.g., using induction), and then conclude using that invariant that the algorithm correctly sorts the array.

(Master Solution is not that precise in)

Hint: Use the invariant “at the moment when the variable i gets incremented to a new value $i = k$ for the first time, the first k elements of the array are sorted in increasing order”.

Inv(k) = At the moment when the variable i gets incremented to a new value $i = k$ for the first time, the first k elements $A[0..k-1]$ are sorted in increasing order

B.C. Let $k=1$ At the moment when the variable i gets incremented to $i=1$ for the first time, the first element $A[0]$ is sorted in increasing order, because a single element is always trivially sorted. Thus **Inv(1)** holds

I.H: Assume **Inv(k)** for some arbitrary but fixed $k \in \{1, \dots, n-1\}$

I.S: We look at the moment when the variable i gets incremented to $i = k$ for the first time By our I.H. we know that $A[0..k-1]$ is sorted

Since $i = k < n$, we start the next iteration

Case $A[i] \geq A[i-1]$. i gets incremented to $i = k+1$ for the first time

Since $A[0..k-1]$ is sorted and $A[\overset{i}{k}] \geq A[\overset{i-1}{k-1}]$, it follows that

the first $k+1$ elements $A[0..k]$ are sorted Thus **Inv($k+1$)** holds.

Case $A[i] < A[i-1]$: The algorithm swaps $A[i]$ repeatedly with its left neighbor $A[i-1]$ as long as its left neighbor is strictly larger, that is, until it is at the correct position. This means that the first $k+1$ elements $A[0..k]$ are sorted. Then i is incremented until $i=k+1$ for the first time. Thus $\text{Inv}(k+1)$ holds.

Termination: Assume that $\text{Inv}(n)$ holds. At termination when the loop ends, i has the value $i=n$. By $\text{Inv}(n)$ the first n elements $A[0..n-1]$ are sorted. Thus the entire array is sorted.

(very annoying and difficult exercise..)

- (b) For the following code snippet, compute the number $T(n)$ of calls to f . Give the exact number of calls. You may use summation signs in your expression.

Algorithm 3

$i \leftarrow 0$ $\Rightarrow$ but our sum starts at $i=1$, as $\log(0)$ is undefined

while $i \leq n$ **do** for $i=0$, no calls happen as $1=k \neq i=0$

$j \leftarrow 1$

while $j^2 \leq i$ **do** $\Leftrightarrow j \leq \lfloor \sqrt{i} \rfloor$ Überlegung: Sei $j=4, i=10$
 $4^2 \neq 10$ deshalb soll $4 \notin \lfloor \sqrt{10} \rfloor$

$j \leftarrow j+1$

$k \leftarrow 1$

while $k \leq i$ **do** $\Leftrightarrow k \leq \lfloor \log i \rfloor + 1$ Überlegung:

$k \leftarrow 2k$

$k \leftarrow k+1$

$f()$

$i \leftarrow i+1$

i	calls	$\log i$
1	1	0
2	2	1
3	2	
4	3	2
5	3	
6	3	

$$\begin{aligned}
 T(n) &= \sum_{i=1}^n \sum_{j=1}^{\lfloor \sqrt{i} \rfloor} \sum_{k=1}^{\lfloor \log i \rfloor + 1} 1 \\
 &= \sum_{i=1}^n \sum_{j=1}^{\lfloor \sqrt{i} \rfloor} (\lfloor \log i \rfloor + 1) \\
 &= \sum_{i=1}^n \lfloor \sqrt{i} \rfloor \cdot (\lfloor \log i \rfloor + 1)
 \end{aligned}$$

- (c) Determine a maximally simplified asymptotic bound in Θ notation for the function $T(n)$ from part (b). You may assume that $n \geq 10$.

obere Schranke: $T(n) \leq \sum_{i=1}^n \sqrt{n} (\log(n)+1) \leq n \sqrt{n} (\log(n)+1) \leq n^{1.5} 2 \log n \leq O(n^{1.5} \log n)$

untere Schranke: $T(n) = \sum_{i=1}^n \lfloor \sqrt{i} \rfloor \cdot (\lfloor \log i \rfloor + 1)$

$\hookrightarrow \lfloor x \rfloor \geq x-1$

$\geq \sum_{i=1}^n (\sqrt{i} - 1) \log(i)$

$\hookrightarrow$ start sum only at $i = \lceil \frac{n}{2} \rceil$

$\geq \sum_{i=\lceil \frac{n}{2} \rceil}^n (\sqrt{i} - 1) \log(i)$

$\hookrightarrow$ lower bound formula from last week

$\geq \frac{1}{2} \left(\sqrt{\frac{n}{2}} - 1 \right) \log\left(\frac{n}{2}\right)$

$\hookrightarrow \log\left(\frac{n}{2}\right) = \log(n) - \log(2)$
 $= \log(n) - 1$

$\geq \frac{1}{2} \left(\sqrt{\frac{n}{2}} - 1 \right) (\log(n) - 1)$

$\hookrightarrow$ we are given that $n \geq 10$.

$\geq \frac{1}{2} \cdot \frac{1}{100} \sqrt{\frac{n}{2}} \cdot \frac{1}{100} \log(n)$

we just pick a constant $= \frac{1}{100}$

$\geq \Omega(n^{1.5} \log n)$

small enough so that we can ignore the "-1"

daraus folgt: $T(n) = \Theta(n^{1.5} \log n)$

Master Theorem

Theorem 1 (Master theorem). Let $a, C > 0$ and $b \geq 0$ be constants and $T : \mathbb{N} \rightarrow \mathbb{R}^+$ a function such that for all even $n \in \mathbb{N}$,

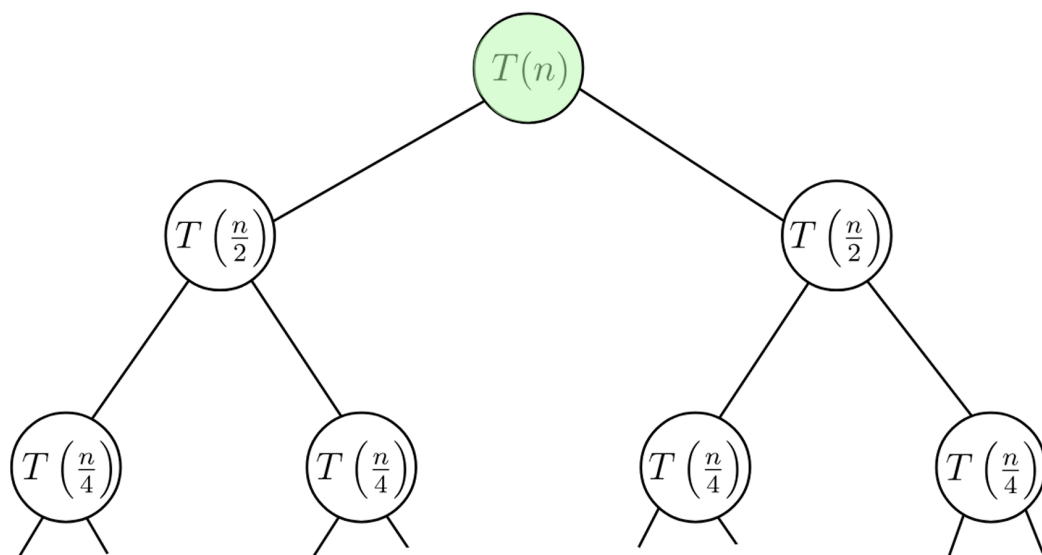
$$T(n) \leq aT(n/2) + Cn^b.$$

Then for all $n = 2^k$, $k \in \mathbb{N}$, the following statements hold

- (i) If $b > \log_2 a$, $T(n) \leq O(n^b)$.
- (ii) If $b = \log_2 a$, $T(n) \leq O(n^{\log_2 a} \cdot \log n) \Leftarrow O(n^b \log n)$
- (iii) If $b < \log_2 a$, $T(n) \leq O(n^{\log_2 a})$.

If the function T is increasing, then the condition $n = 2^k$ can be dropped.

Graph für $a=2$.

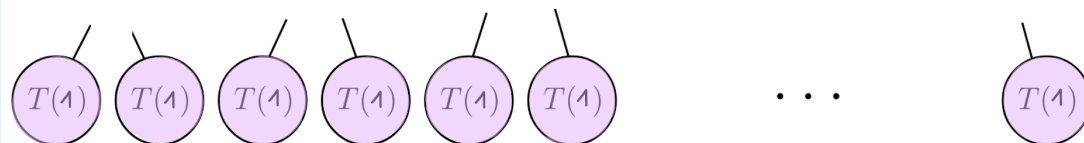


Level 0: $\underbrace{n^b \cdot C}_{\text{Arbeit pro Knoten}}$

Level 1: $\underbrace{a}_{\text{Anzahl Knoten}} \cdot \underbrace{\left(\frac{n}{2}\right)^b \cdot C}_{\text{Arbeit pro Knoten}}$

Level 2: $\underbrace{a^2}_{\text{Anzahl Knoten}} \cdot \underbrace{\left(\frac{n}{2^2}\right)^b \cdot C}_{\text{Arbeit pro Knoten}}$

Level i: $\underbrace{a^i}_{\text{Anzahl Knoten}} \cdot \underbrace{\left(\frac{n}{2^i}\right)^b \cdot C}_{\text{Arbeit pro Knoten}}$



Level $\log_2 n$:
 $\underbrace{a^{\log_2 n}}_{\text{Anzahl Knoten}} \cdot \underbrace{1 \cdot C}_{\text{Arbeit pro Knoten}} = n^{\log_2 a} \cdot C$

Case $b > \log_2 a$: oberstes Level dominiert, andere Level vernachlässigbar $\Rightarrow O(n^b)$

Case $b = \log_2 a$: alle Level haben genau gleich viel Arbeit $\Rightarrow O(n^b \cdot \underbrace{\log n}_{\text{Anzahl Level}})$

Case $b < \log_2 a$: unterstes Level dominiert, andere Level vernachlässigbar $\Rightarrow O(n^{\log_2 a})$

Beweis:

Potenzgesetz

Distributivgesetz

Potenzgesetz

$$\text{Totale Arbeit: } \sum_{i=0}^{\log_2 n} a^i \left(\frac{n}{2^i}\right)^b \xrightarrow{\text{Potenzgesetz}} \sum_{i=0}^{\log_2 n} a^i \frac{n^b}{2^{i \cdot b}} \xrightarrow{\text{Distributivgesetz}} n^b \sum_{i=0}^{\log_2 n} \frac{a^i}{2^{i \cdot b}} \xrightarrow{\text{Potenzgesetz}} n^b \sum_{i=0}^{\log_2 n} \left(\frac{a}{2^b}\right)^i = \dots$$

Case $b > \log_2 a$: $b > \log_2 a \Leftrightarrow 2^b > a \Leftrightarrow \frac{a}{2^b} < 1$

$$n^b \sum_{i=0}^{\log_2 n} \left(\frac{a}{2^b}\right)^i \leq n^b \sum_{i=0}^{\infty} \left(\frac{a}{2^b}\right)^i = n^b \underbrace{\frac{1}{1 - \frac{a}{2^b}}}_{\text{Konstante}} \leq O(n^b)$$

Geometrische Reihe:

$$\sum_{i=0}^{\infty} r^i = \frac{1}{1-r} \quad \text{falls } r < 1$$

Case $b = \log_2 a$: $b = \log_2 a \Leftrightarrow 2^b = a$

$$n^b \sum_{i=0}^{\log_2 n} \left(\frac{a}{2^b}\right)^i \underset{2^b = a}{=} n^b \sum_{i=0}^{\log_2 n} \underbrace{\left(\frac{a}{2^b}\right)^i}_{=1} = n^b \sum_{i=0}^{\log_2 n} 1 \leq O(n^b \cdot \log n)$$

Case $b < \log_2 a$: $b < \log_2 a \Leftrightarrow 2^b < a \Leftrightarrow \frac{a}{2^b} > 1$

$$n^b \sum_{i=0}^{\log_2 n} \left(\frac{a}{2^b}\right)^i = n^b \frac{\left(\frac{a}{2^b}\right)^{\log_2(n)+1} - 1}{\left(\frac{a}{2^b}\right) - 1}$$

$$\leq O\left(n^b \cdot \left(\frac{a}{2^b}\right)^{\log_2 n}\right)$$

$$\leq O\left(n^b \cdot \frac{a^{\log_2 n}}{2^{b \log_2 n}}\right)$$

$$\leq O\left(n^b \cdot \frac{a^{\log_2 n}}{2^{\log_2(n^b)}}\right)$$

$$\leq O\left(n^b \cdot \frac{a^{\log_2 n}}{n^b}\right)$$

$$\leq O(a^{\log_2 n})$$

$$\leq O(n^{\log_2 a})$$

Geometrische Reihe:

$$\sum_{i=0}^k r^i = \frac{r^{k+1} - 1}{r - 1} \quad \text{falls } r > 1$$

↳ Konstanten weglassen

↳ Potenzgesetz

↳ log Gesetz

↳ $2^{\log_2 x} = x$ ↳ n^b kürzen↳ log Gesetz: $x^{\log y} = y^{\log x}$

Aufgaben:

Let $T(1) = 3, T(n) = 2T(n/2) + n^2$ for $n > 1$. Using the master theorem, show that

$$T(n) \leq O(n^2).$$

$$a = 2, b = 2.$$

$\log_2(a) = \log_2(2) = 1 < 2 = b \Rightarrow$ Fall oberstes Level dominiert, andere Level vernachlässigbar

Master Theorem

$$\Rightarrow T(n) \leq O(n^b) = O(n^2)$$

Algorithm 1

function $A(n)$

$i \leftarrow 0$

while $i < n^2$ do

$j \leftarrow n$

while $j > 0$ do

$f()$

$f()$

$j \leftarrow j - 1$

$i \leftarrow i + 1$

$k \leftarrow \lfloor \frac{n}{2} \rfloor$

for $l = 0 \dots 3$ do

if $k > 0$ then

$A(k)$

$A(k)$

Wir zählen den loop rückwärts durch

$$\text{Number of calls} = \sum_{i=0}^{n^2-1} \sum_{j=1}^n 2 + \sum_{l=0}^3 2 A(\lfloor \frac{n}{2} \rfloor)$$

$$= \sum_{i=0}^{n^2-1} 2n + 4 \cdot 2 A(\lfloor \frac{n}{2} \rfloor)$$

$$= (n^2 - 1 + 1) \cdot 2n + 8 A(\lfloor \frac{n}{2} \rfloor)$$

$$= 2n^3 + 8 A(\lfloor \frac{n}{2} \rfloor)$$

$$\hookrightarrow \lfloor \frac{n}{2} \rfloor \leq \frac{n}{2}$$

$$\leq 2n^3 + 8 A(\frac{n}{2})$$

$$b=3 \quad \hat{a}=8$$

$$\log_2 a = \log_2 8 = 3 = b$$

Master Theorem

$$\Rightarrow A(n) \leq O(n^3 \log n)$$

Algorithm 2

function $C(n)$

if $n \leq 1$ then

return 0

$j \leftarrow 1$

while $\sqrt{j} \leq n^3$ do

$$\Leftrightarrow j \leq n^6$$

$f()$

$j \leftarrow j + 1$

$k \leftarrow \lfloor n/2 \rfloor$

$C(k)$

$C(k)$

$C(k)$

$$\text{Number of calls} = \sum_{j=1}^{n^6} 1 + 3 C(\lfloor \frac{n}{2} \rfloor)$$

$$= n^6 + 3 C(\lfloor \frac{n}{2} \rfloor)$$

$$\hookrightarrow \lfloor \frac{n}{2} \rfloor \leq \frac{n}{2}$$

$$\leq n^6 + 3 C(\frac{n}{2})$$

$$b=6 \quad \hat{a}=3$$

$$3 < 2^6$$

$$\Rightarrow \log_2(3) < 6$$

$$\Rightarrow \log_2(a) < b$$

Master Theorem

$$\Rightarrow C(n) \leq O(n^6)$$

Assume that the call $f()$ does not make any recursive calls. Consider the mutually recursive algorithms $g(n)$ and $h(n)$ below, where $n \geq 1$ is a power of 2.

Let $G(n)$ be the number of function calls to f that the function $g(n)$ makes.

Let $H(n)$ be the number of function calls to f that the function $h(n)$ makes.

Give the asymptotic bound in O -Notation for $G(n)$ and for $H(n)$.

Algorithm 2

function $g(n)$

$i \leftarrow 1$

while $i \leq n$ **do**

$f()$

$i \leftarrow i + 1$

$h(n/2)$

$h(n/2)$

function $h(n)$

if $n \leq 1$ **then**

return

$i \leftarrow 1$

while $i - 1 < n^2$ **do**

$f()$

$i \leftarrow i + 1$

$j \leftarrow 0$

while $j \leq 6$ **do**

$g(n)$

$j \leftarrow j + 2$

$$(i) \quad G(n) = \left(\sum_{i=1}^n 1 \right) + 2 H\left(\frac{n}{2}\right) = n + 2 H\left(\frac{n}{2}\right)$$

$$(ii) \quad H(n) = \left(\sum_{i=1}^{n^2} 1 \right) + \sum_{j=0}^3 G(n) = n^2 + 4 G(n)$$

$$(i) \text{ in } (ii) \text{ einsetzen: } H(n) = n^2 + 4 G(n)$$

$$= n^2 + 4 \left(n + 2 H\left(\frac{n}{2}\right) \right)$$

$$= n^2 + 4n + 8 H\left(\frac{n}{2}\right)$$

$$\leq n^2 + 4n^2 + 8 H\left(\frac{n}{2}\right)$$

$$= 5n^2 + 8 H\left(\frac{n}{2}\right)$$

Master Theorem

$$\leq O(n^3)$$

$$\log a = \log 8 = 3 > 2 = b$$

$$(ii) \text{ in } (i) \text{ einsetzen: } G(n) = n + 2 H\left(\frac{n}{2}\right)$$

$$= n + 2 \left(\left(\frac{n}{2}\right)^2 + 4 G\left(\frac{n}{2}\right) \right)$$

$$= n + \frac{n^2}{2} + 8 G\left(\frac{n}{2}\right)$$

$$\leq 2n^2 + 8 G\left(\frac{n}{2}\right)$$

Master Theorem

$$\leq O(n^3)$$

$$\log a = \log 8 = 3 > 2 = b$$

Sortieren

QUICKSORT($A[1..n], l, r$)

1 **if** $l < r$ **then**

2 $k \leftarrow \text{Aufteilen}(A, l, r)$

3 QuickSort($A, l, k - 1$)

4 QuickSort($A, k + 1, r$)

▷ Teile $A[l..r]$ in zwei Gruppen auf

▷ Sortiere linke Gruppe

▷ Sortiere rechte Gruppe

AUFTEILEN($A[1..n], l, r$)

1 $i \leftarrow l$

2 $j \leftarrow r - 1$

3 $p \leftarrow A[r]$

4 **repeat**

5 **while** $i < r$ **and** $A[i] \leq p$ **do**

6 $i \leftarrow i + 1$

7 **while** $j > l$ **and** $A[j] > p$ **do**

8 $j \leftarrow j - 1$

9 **if** $i < j$ **then**

10 Vertausche $A[i]$ und $A[j]$

11 **until** $i \geq j$

12 Vertausche $A[i]$ und $A[r]$

13 **return** i

▷ Index für die linke Gruppe

▷ Index für die rechte Gruppe

▷ Pivotelement

▷ Suche nächstes Element für rechte Gruppe

▷ Suche nächstes Element für linke Gruppe

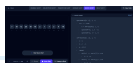
▷ Vertausche Elemente

▷ Schleife endet, wenn sich i und j „treffen“

▷ Pivotelement an die richtige Stelle tauschen

▷ Gib Position des Pivotelements zurück

Bsp



Fall $\log a = \log 2 = 1 = b$

Best Case

$$T(n) \leq \underbrace{2}_{\text{Rekursion}} \underbrace{T\left(\frac{n}{2}\right)}_{\text{Aufteilen}} + \underbrace{c}_{\text{Aufteilen}} n \quad \xRightarrow{\text{Master Theorem}} O(n \log n)$$

Average Case:
(zufälliges Pivot)

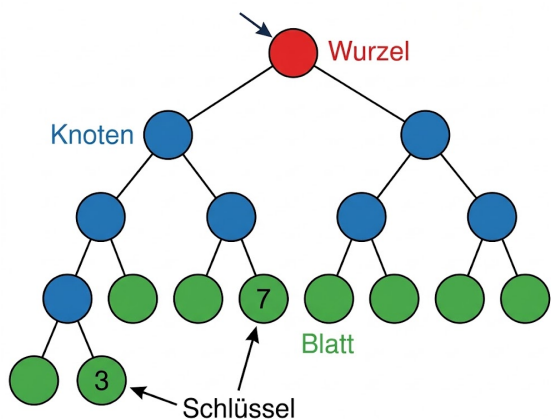
$$\underbrace{E[T(n)]}_{\text{Erwartete Laufzeit}} \leq O(n \log n)$$

Worst Case:

$$T(n) \leq \underbrace{T(n-1)}_{\text{Rekursion}} + \underbrace{c}_{\text{Aufteilen}} n \quad \xRightarrow{\text{Teleskopieren}} O(n^2)$$

Max-Heap

Vollständiger Binärbaum:

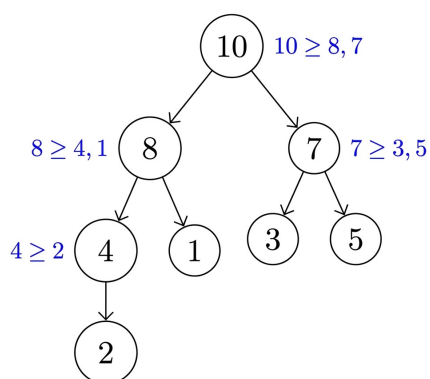


- Alle Knoten, die nicht in den beiden untersten Levels sind, haben genau 2 Kinder.
- Knoten im zweituntersten Level haben 0, 1, oder 2 Kinder
- Knoten im untersten Level haben 0 Kinder
- unterstes Level ist von links nach rechts aufgefüllt

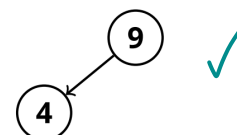
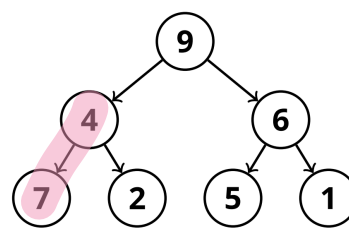
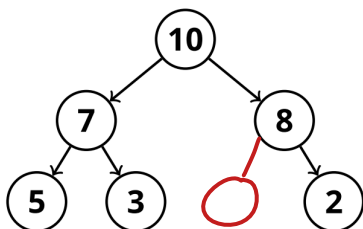
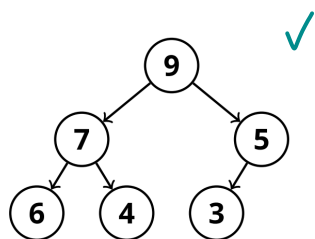
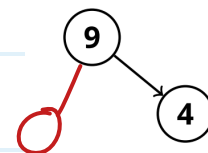
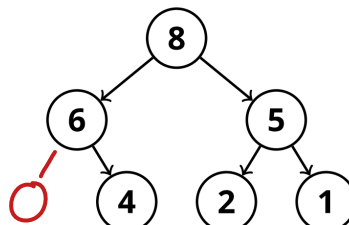
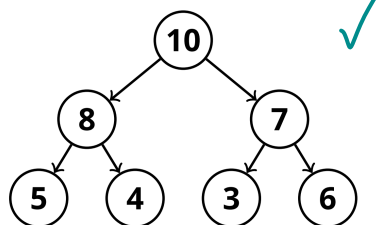
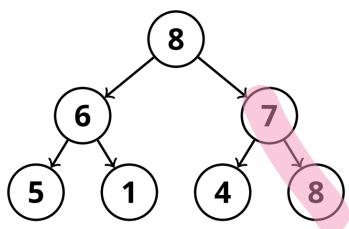
(Wurzel und Blätter sind auch Knoten)

Heap Bedingung:

- Schlüssel des Knotens $\geq$ Schlüssel seiner Kinder



Aufgabe: Entscheide ob folgende Bäume korrekte Max-Heaps sind oder nicht.



ExtractMax: in $O(\log n)$

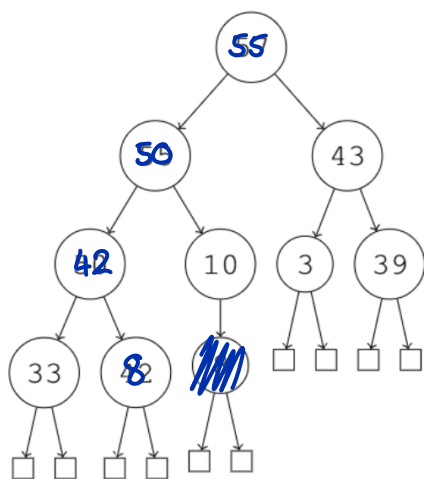
1.) Wurzel (=Max) entfernen

2.) Letztes Blatt an die frei gewordene Stelle der Wurzel tauschen

3.) Die (neue) Wurzel versickern lassen, d.h. so lange mit dem grössten Kind tauschen, bis die Heap-Bedingung wieder gilt.

i) Extract the maximum of the following **max-heap** and do all necessary operations to restore the heap condition (it suffices to only draw the final tree).

2 min an Prüfung

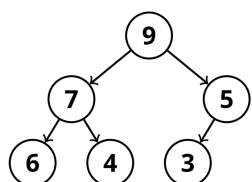


Einfügen: in $O(\log n)$

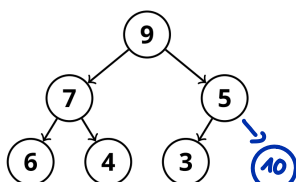
1.) setze den neuen Knoten an die nächste freie Stelle

2.) tausche ihn mit den Eltern Knoten bis die Heap-Bedingung wieder gilt.

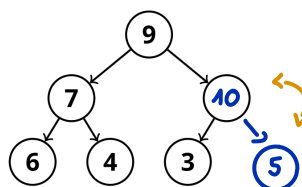
Aufgabe: Füge den Schlüssel 10 ein.



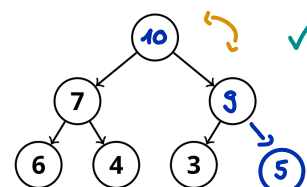
→



→



→



HEAPSORT($A[1..n]$)

1 $H \leftarrow \text{Heapify}(A)$ $\} O(n)$

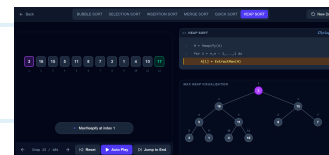
2 **for** $i \leftarrow n, n-1, \dots, 1$ **do**

3 $A[i] \leftarrow \text{ExtractMax}(H)$ $O(\log n)$

▷ Wandle Array in Heap um.

▷ Entferne Elemente aus Heap

$\} O(n \log n)$



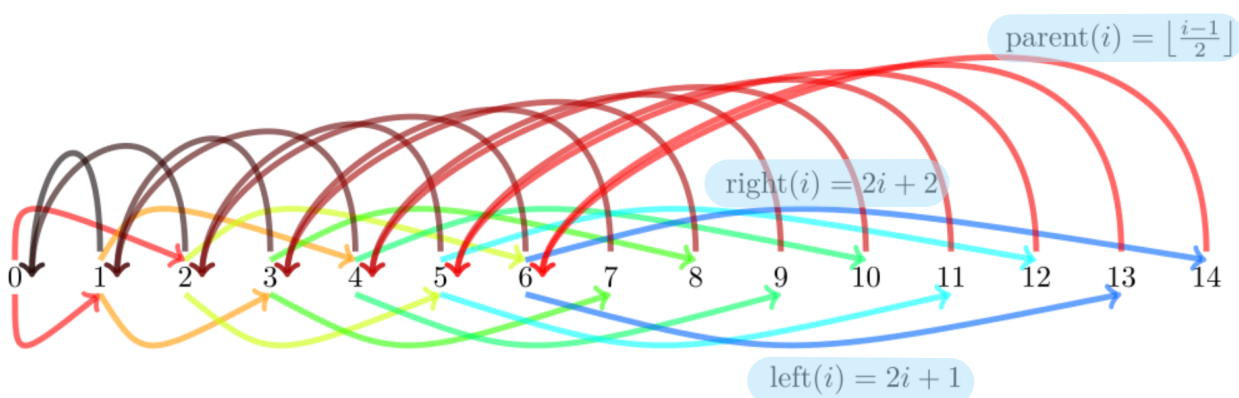
Laufzeit: $O(n \log n)$

Heapify: möglich in $O(n)$

1.) Fülle die Elemente des Arrays in einen vollständigen Binärbaum

2.) Versichere alle Knoten in aufsteigender Reihenfolge der Levels

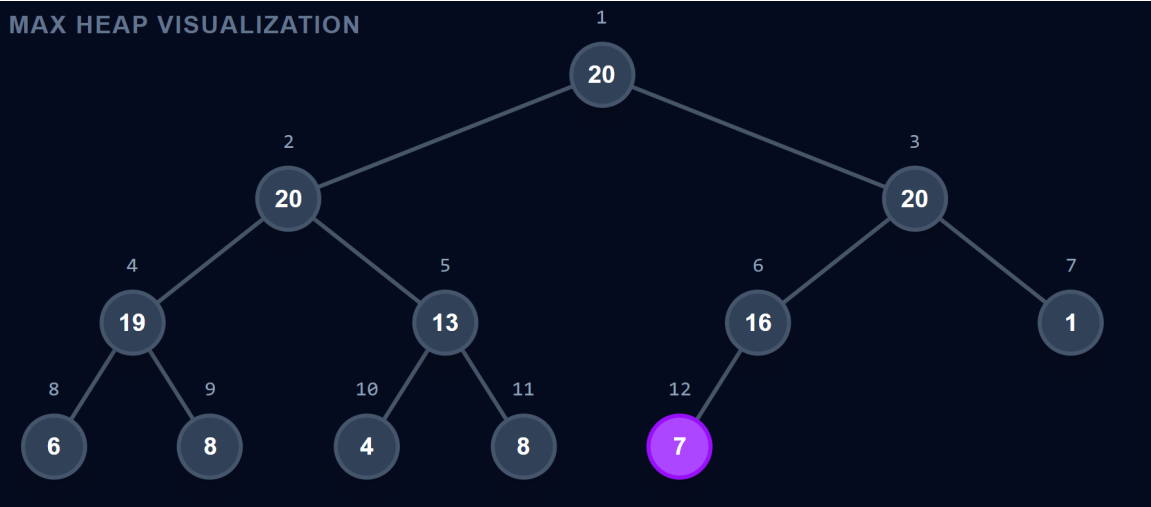
Darstellung des Max-Heaps als Array:



Achtung:
In Vorlesung
 $2i$ und $2i+1$

$\Rightarrow$ In-Place

MAX HEAP VISUALIZATION



$$A = [a_1 \ a_2 \ a_3 \ \dots \ a_n]$$

Heapify

tausche Wurzel und letztes Blatt

$$[a_n \ a_2 \ a_3 \ \dots \ a_1]$$

versichern

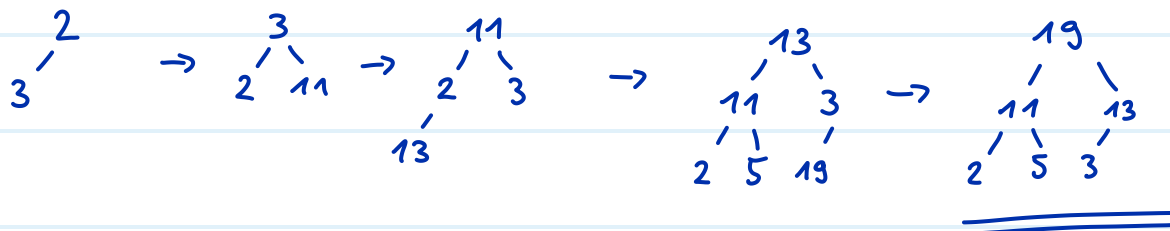
$$[a_2 \ a_n \ a_3 \ \dots \ a_1]$$

Heap

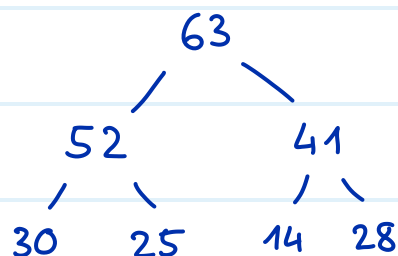
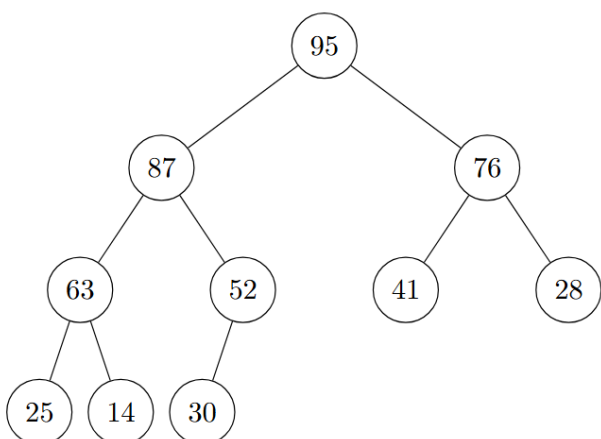
$\lnv(i)$ = grössten, Elemente sind sortiert

Aufgaben:

- iii) Draw the **max-heap** obtained from inserting the keys 2, 3, 11, 13, 5, 19 in this order into an empty heap. It suffices to only draw the final heap. If there are several solutions, it is enough to draw only one of them.



Draw the Heap after ExtractMax is called three times



Beispiel für nicht vergleichbasiertes Sortieren:

BUCKET-SORT($A[1..n]$)

▷ alle Einträge in $\{1, \dots, n\}$

1 $B[1..n] \leftarrow [0, 0, \dots, 0]$

funktioniert sonst nicht

2 **for** $j \leftarrow 1, 2, \dots, n$ **do**

3 $B[A[j]] \leftarrow B[A[j]] + 1$

▷ Zähle in $B[i]$, wie oft i vorkommt

4 $k \leftarrow 1$

5 **for** $i \leftarrow 1, 2, \dots, n$ **do**

6 $A[k, \dots, k + B[i] - 1] \leftarrow [i, i, \dots, i]$

▷ Schreibe $B[i]$ -mal den Wert i in A

7 $k \leftarrow k + B[i]$

▷ A ist bis Position $k + B[i] - 1$ gefüllt.

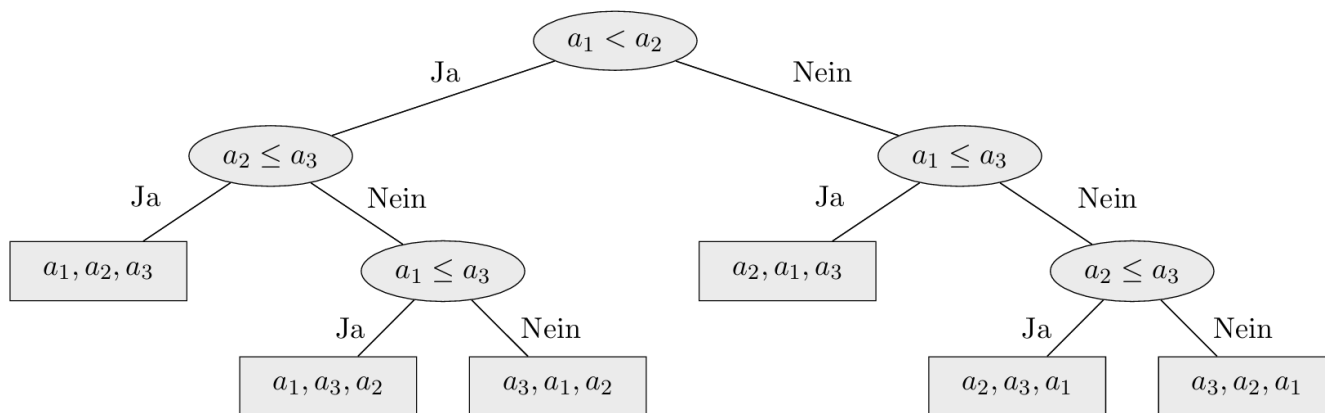
Laufzeit: $\Theta(n)$

Bsp: $A = [3 \ 4 \ 4 \ 6 \ 5 \ 1 \ 2 \ 6]$

$B = [1 \ 1 \ 1 \ 2 \ 1 \ 2 \ 0 \ 0]$

Ausgabe = $[1 \ 2 \ 3 \ 4 \ 4 \ 5 \ 6 \ 6]$

Entscheidungsbaum:



■ Abb. 2.5 Beispiel Entscheidungsbaum mit Array $[a_1, a_2, a_3]$

Wie viele Blätter (= mögliche Ausgaben = Anzahl Permutationen)? $\rightarrow n(n-1)(n-2)\dots 1 = n!$

Auf Höhe h gibt es genau 2^h viele Knoten

Wie hoch muss der Baum mindestens sein, damit er $n!$ viele Blätter haben kann?

$$\begin{aligned}
 2^h &\geq n! \\
 \Rightarrow 2^h &\geq \left(\frac{n}{2}\right)^{\frac{n}{2}} && \left. \begin{array}{l} \hookrightarrow n! \geq \left(\frac{n}{2}\right)^{\frac{n}{2}} \\ \hookrightarrow \text{auf beiden Seiten log nehmen} \\ \hookrightarrow \text{log Gesetz} \end{array} \right. \\
 \Rightarrow h &\geq \log\left(\left(\frac{n}{2}\right)^{\frac{n}{2}}\right) \\
 \Rightarrow h &\geq \frac{n}{2} \cdot \log\left(\frac{n}{2}\right) \\
 \Rightarrow h &\geq \Omega(n \log n)
 \end{aligned}$$

Alternativ:

1 Vergleich $\Leftrightarrow$ wir können genau 2 Permutationen unterscheiden

2 Vergleiche $\Leftrightarrow$ wir können genau 4 Permutationen unterscheiden

i Vergleiche $\Leftrightarrow$ wir können genau 2^i Permutationen unterscheiden

Wir müssen $n!$ Permutationen unterscheiden

$\Rightarrow$ wir brauchen $\log(n!) \geq \Omega(n \log n)$ viele Vergleiche

Abstrakte Datentypen

Abstraktion: bedeutet immer das Vereinfachen/Weglassen bestimmter Details eines Problems

→ Bsp Strassenkarten, Zahlen, Programmiersprachen, Logik, ...

Abstrakter Datentyp: beschreibt was mit den Daten passieren soll, ohne im Detail festzulegen wie genau es passieren soll. (⇒ Interfaces in EProg)

ADT Liste:

- $\text{insert}(k, L)$: Fügt den Schlüssel k am Ende der Liste L ein.
- $\text{get}(i, L)$: Gibt den Speicherort des i -ten Objekts in L zurück.
- $\text{delete}(k, L)$: Entfernt den Schlüssel k aus der Liste L .
- $\text{insertAfter}(k, k', L)$: Fügt den Schlüssel k' hinter dem Schlüssel k in die Liste L ein.

Drei Implementierungen des ADT Liste:

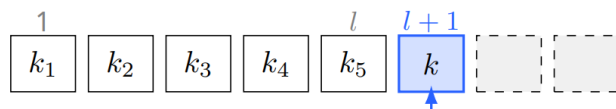
- **Arrays:** Fester, zusammenhängender Speicherbereich mit bekannter Maximalgrösse
- **Einfach verkettete Liste:** ein Element hat immer einen Pointer zum nächsten
- **Zweifach verkettete Liste:** ein Element hat immer einen Pointer zum vorherigen und zum nächsten

Laufzeiten:

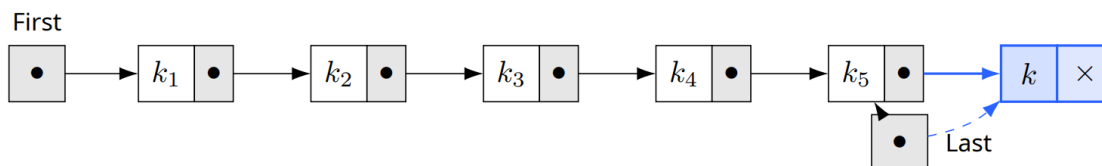
	Array	einf. verlinkte Liste	dopp. verlinkte Liste
$\text{insert}(k, L)$	$O(1)$	$O(1)$	$O(1)$
$\text{get}(i, L)$	$O(1)$	$O(i) \leq O(\ell)$	$O(i) \leq O(\ell)$
$\text{insertAfter}(k, k', L)$	$O(\ell)$	$O(1)$	$O(1)$
$\text{delete}(k, L)$	$O(\ell)$	$O(\ell)$	$O(1)$

1. insert(k, L) (Am Ende einfügen)

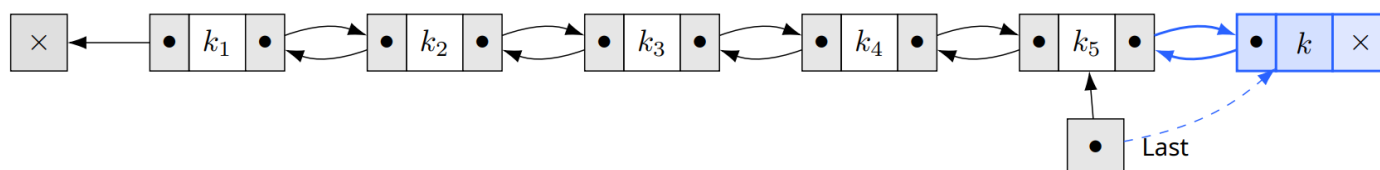
1.1 Array - Laufzeit: $O(1)$



1.2 Einfach verkettete Liste - Laufzeit: $O(1)$

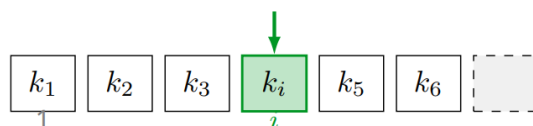


1.3 Doppelt verkettete Liste - Laufzeit: $O(1)$



2. get(i, L) (Speicherort des i -ten Elements abfragen)

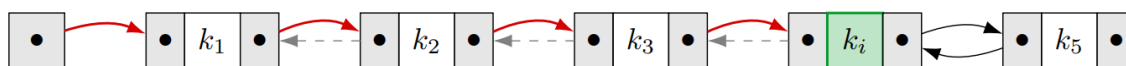
2.1 Array - Laufzeit: $O(1)$



2.2 Einfach verkettete Liste - Laufzeit: $O(l)$

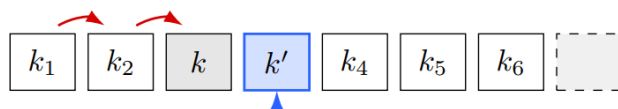


2.3 Doppelt verkettete Liste - Laufzeit: $O(l)$

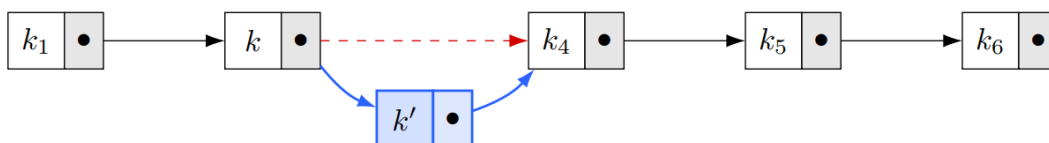


3. insertAfter(k, k', L) (Einfügen nach bekanntem Knoten k)

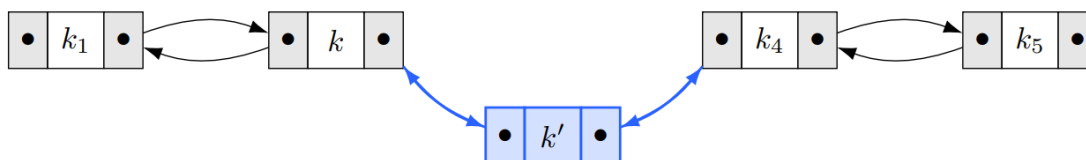
3.1 Array – Laufzeit: $O(l)$



3.2 Einfach verkettete Liste – Laufzeit: $O(1)$

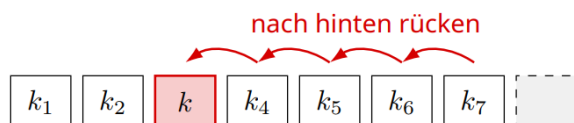


3.3 Doppelt verkettete Liste – Laufzeit: $O(1)$

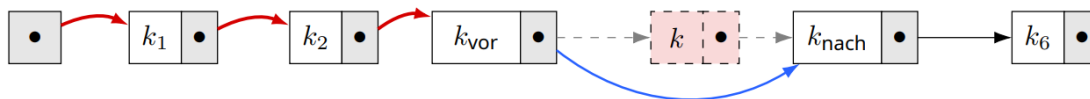


4. delete(k, L) (Schlüssel k aus Liste entfernen)

4.1 Array – Laufzeit: $O(l)$



4.2 Einfach verkettete Liste – Laufzeit: $O(l)$



4.3 Doppelt verkettete Liste – Laufzeit: $O(1)$

